Term Rewriting with Genetic Programming

or, unfortunately, writing a lot of code that attempts to do term rewriting with genetic programming, but doesn’t actually work

Rewrite simplification of Prop

Simplification: representing a term in an equivalent, smaller form - what we did in Ex. 2.

Could apply this to other logics (or programs) - not that I did.

Prop

Familiar, but not parametrized on variable index:

```
data Prop = Top | Bottom | Letter Int
          | Not Prop
          | And Prop Prop
          | Or Prop Prop
          | Implies Prop Prop
```

Term rewriting

Goal is to reduce a large proposition to something that might be easier to do other computations on - fewer variables and clauses

Also some ideal cases: reduce to T or F, or clauses where each literal or conjugate appears only once, etc

Propositional term rewriting: approaches

- Replace the proposition with an equivalent proposition - structural, syntactic rules, semantic.
- Apply a sequence of one-layer transformations.
- Apply transformations in a recursive fashion (e. g. bottom up).
- Measure the proposition somehow and decide how to proceed.

Rewrites

```
data Rewrite =
          Prim(Prop -> Prop) String
        | Pred (Prop -> Bool) Rewrite String
        | SizePred  (Prop -> Int) Int Rewrite String
        | EqPred
              (Prop -> Prop -> Bool) ((Prop -> Prop -> Bool) -> Prop -> Prop)
              String String
        | Algo ((Prop -> Prop) -> Prop -> Prop) Rewrite String
        | Converge (Prop -> Prop -> Bool) Int Rewrite String
        | Sequence [Rewrite]
```

Idea: search the space of rewrite ASTs

Take the building blocks for “good general approaches” and test them against random propositions.

AST is easy to manipulate and change, so should work well for a genetic algorithm.

Genetic Algorithms

Start with random attempts at a solution.

Then, repeatedly:

> Use a fitness function to rank the attempts.
>
> Select the best attempts and mutate them.
>
> Select pairwise combinations of the mutants, and combine them to make new solutions.
>
> Use the new solutions as the next starting

Genetic Programming

The “solution” is itself a program.

In this case: an AST for rewriting strategies.

The fitness function is based on how well the program does at rewriting the propositions:

speed (real time), and size reduction

Example rewrite primitives

Some may shrink the proposition a bit:

```
deMorganPull :: Prop -> Prop
deMorganPull p = case p of
    Or (Not p) (Not q)    -> Not (And p q)
    And (Not p) (Not q)  -> Not (Or p q)
    p                               -> p
```

Example rewrite primitives

Some may not change the proposition size

```
assocRight :: Prop -> Prop
assocRight p = case p of
    And (And p q) r -> And p (And q r)
    Or  (Or p q)  r -> Or  p (Or q r)
    p -> p
```

Example rewrite primitives

Or may grow it:

```
distribute :: Prop -> Prop
distribute p = case p of
    And p (Or q r)              -> Or  (And p q) (And p r)
    And (Or q r) p              -> Or  (And p q) (And p r)
    Or p (And q r)              -> And (Or p q)  (Or p r)
    Or (And q r) p              -> And (Or p q)  (Or p r)
    Implies p (Implies q r)     -> Implies (Implies p q) (Implies q r)

    p                         -> p
```

Example rewrite primitives

Some may use semantic information (since the primitives are just Haskell functions)

```
elimTautology :: Prop -> Prop
elimTautology p
  | taut = Top
  | unsat = Bottom
  | otherwise = p
    where (taut, unsat) = satInfo p -- computes both at once
```

Example “EqPred” primitives

```
elimConjugate :: (Prop -> Prop -> Bool) -> Prop -> Prop
elimConjugate eq p = case p of
    And p q     | eq p (Not q)  -> Bottom
    Or  p q     | eq p (Not q)  -> Top
    p                           -> p
```

Example rewrite algorithms

Top down.

Bottom up.

Apply until convergence.

Use a predicate to select a specific subtree and recurse, such as by number of variables or size.

Go depth-first along the AST, picking a specific subtree at each level with a predicate

Rewrites from Ex 2 student solutions

```
until convergence or at most 10 iterations {
    apply applied top down {
        rewrite(shrinking identities)
        rewrite(eliminate duplicate clauses)
           using (structural equality)
        rewrite(push deMorgan involution)
    }
    rewrite(simplify via CNF)
} -- by Ted Cooper
```

Rewrites from Ex 2 student solutions

```
apply top down {
    rewrite(factor) using (structural equality)
}
apply bottom up {
    rewrite(shrinking identities)
}
rewrite(simplify via CNF) -- by Kendall Stewart
```

Random propositions

genProp :: [Double] -> Int -> Int -> IO Prop

genProp connectives max_vars size

connectives is something like

[1,1,1,1] -- all connectives

[1,1,0,0] -- nothing but Not and And

[2,0.5,0.25,3] -- lots of Implies, a bit of Not,

Random proposition examples

```
./RandomProp 1 1 1 1 10 100  -v
(A ∨ B ⇒ ¬¬((¬((B ∧ B) ⇒ C ⇒ ¬(A ∧ ((B ⇒ C) ∨ A)) ⇒ (A ∧ C) ⇒ (A ∨
A)) ∨ C ∨ C ⇒ B) ∨ (((((D ⇒ E) ⇒ C) ∧ E) ⇒ (E ∧ (B ⇒ F) ∧ (D ∨ B ⇒
D))) ⇒ (¬(A ⇒ D) ∨ (¬(¬(G ⇒ A) ∨ (H ∨ A ⇒ ¬E)) ∧ A))))) ∨ F ∨ E ∨ (F ⇒
¬(B ∧ (G ∨ A)))
Size:   87
Vars:   8
Taut?   True
Sat?    True
Absurd? False
```

Random proposition examples

```
./RandomProp 1 1 0 0 30 200  -v
```

¬(A ∧ ¬A ∧ ¬¬(B ∧ ¬B ∧ B) ∧ ¬(¬C ∧ ¬(¬¬¬¬(A ∧ ¬¬(B ∧ B ∧ C ∧ ¬(B
∧ D) ∧ A) ∧ D) ∧ ¬(¬(A ∧ E) ∧ F ∧ ¬(C ∧ E ∧ D))) ∧ ¬¬¬¬(¬¬G ∧ G))
∧ ¬(¬(¬¬(¬¬¬(¬(¬E ∧ E ∧ ¬¬(C ∧ G ∧ E)) ∧ E) ∧ H ∧ F) ∧ G) ∧ ¬¬¬(¬¬
(G ∧ F) ∧ D ∧ ¬(C ∧ G) ∧ A)) ∧ ¬¬(¬¬(¬A ∧ B ∧ G ∧ C ∧ ¬¬¬¬B ∧ F
∧ E) ∧ C ∧ ¬¬¬(I ∧ E)) ∧ ¬¬¬¬I ∧ ¬(¬¬(¬(¬¬(H ∧ B) ∧ C) ∧ ¬¬(E ∧ G))
∧ ¬(F ∧ D ∧ J ∧ B)) ∧ ¬¬I ∧ ¬(J ∧ G))

Size: 179

Vars: 10

Taut?   True

Sat?   True

Absurd? False

Statistics of distributions of Prop

```
./PropStats 1 1 1 1 15 200  100
Average Size:   181.75
Average Vars:   11.77
Average Taut?   0.15
Average Sat?    0.89
Average Absurd? 0.11
```

```
./PropStats 1 1 0 0 15 200  100
Average Size:   175.39
Average Vars:   10.48
Average Taut?   0.21
Average Sat?    0.63
Average Absurd? 0.37
```

Random rewrites

Given a size and a pool of primitive functions, randomize a rewrite AST.

This makes some silly programs!

Random rewrite examples

```
until convergence or at most 119 iterations {
    if number of possible assignments is less than 62084 {
        rewrite(Not from contradicting Implies) using (bidirectional implication)
    }
    rewrite(commutivity of And/Or/Implies)
    rewrite(simple identities of Not)
    rewrite(commutivity of And/Or/Implies)
    until convergence or at most 35 iterations {
        rewrite(simple shrinking identities)
    }
}
```

Random rewrite examples

```
apply bottom-up {
    if number of possible assignments is less than 18072 {
        rewrite(negation normal form)
    }
    rewrite(commutivity of And/Or/Implies)
    rewrite(elimination of conjugate pair) using (bidirectional implication)
    rewrite(non-recursive structural equality)
    rewrite(simple identities of Implies)
    rewrite(collapsing double distributed And/Or) using (identical truth tables with union of vars)
}
apply top-down {
    until convergence or at most 249 iterations {
        rewrite(left associativity of And/Or)
    }
}
```

Semantic preservation of rewrites

TODO

Some disappointing results:

```
./RandomPair 1 1 1 1 10 10 10
((A ⇒ A) ∨ B ∨ B ⇒ ¬C) ⇒ B
```

```
if number of possible assignments is less than 59925 {
    until convergence or at most 35 iterations {
        rewrite(simple identities of Implies)
    }
}
if proposition size is less than 9160 {
    rewrite(push Not deeper with deMorgan involution)
}
if proposition size is less than 14459 {
    rewrite(left associativity of And/Or)
}
if number of possible assignments is less than 5330 {
    rewrite(material implication to Implies)
}
rewrite(simple identities of And)
```

```
((A ⇒ A) ∨ B ∨ B ⇒ ¬C) ⇒
B
Size(p):   11
Vars(p):   3
Size(p'):  11
Vars(p'):  3
Equal?     True
```

Some disappointing results

```
./RandomPair 1 1 1 1 10 10 10
(A ⇒ (¬B ∨ B)) ∨ (A ∧ C)
apply bottom-up {
    until convergence or at most 252 iterations {
        rewrite(simple identities of Not)
    }
    if number of possible assignments is less than 38842 {
        rewrite(simple identities of Or)
    }
    ….
Stack space overflow: current size 8388608 bytes.
Use `+RTS -Ksize -RTS' to increase it.
```

Genetic Algorithm Instances

```
--   FitnessFn     :: time   -> p    -> p'   -> score
type FitnessFn      = Double -> Prop -> Prop -> Double
--   SelectionFn   :: score  -> probability of duplication
type SelectionFn= [(Double, Rewrite)] -> [(Double, Rewrite)]
--   MutationFn    :: p      -> rw      -> mutated rw
type MutationFn     = Double -> Rewrite -> RandM Rewrite
--   CrossoverFn   :: p      -> parent 1 -> parent 2 -> child rw
type CrossoverFn    = Double -> Rewrite -> Rewrite -> RandM Rewrite
```

and a bunch of other parameters...

Am I done yet?

I expect to run out of time by about now.

If I did, I can cut a long story short: *it didn't work*.

Many lines of code later...

Assembled a bunch of primitives - still hadn’t done recombination, but wanted to get something up and running to actually test in whole (finally).

But then….

Emergent thunk leak!

I had tested all of the parts of the algorithm independently (albeit ad hoc), but when I ran the whole thing, I got a stack overflow.

It wasn’t clear where it came from.

I started messing around with GHC’s profiler, which gives a lot of information - but no clear solution.

Some partial solutions….

Program sometimes didn’t terminate, even when running within a timeout wrapper function.

This function had worked fine on simpler nonterminating programs (e.g. x = x)...

Eliminating one ridiculous cost center usually just revealed another.

Made things more strict - ! and $! voodoo.

I’m a fool

Turned on optimizations (-O2) and the program runs fine....

Seems bad (non-portable, unreliable) to rely on optimizations for program behavior, but I just couldn’t get it to run without optimization.

Still… moral of the story: trust all-knowing GHC.

(Maybe I could with more ! and $!).

But not a total fool...

When I went back to code that I had corrected or removed in earlier attempts to fix the leak,

old versions of code had thunk leaks, even with optimization.

Some of the strictness voodoo was necessary!

The program in action….
wait, I’m still a fool

Missing functionality - recombinations, plus many algorithms eliminated as they caused space leaks (e.g. bottomUp, topDown) - even though they worked fine on their own.

Laziness made debugging challenging.

I guess I need to use a lot of Debug.Trace or write property/unit tests?

Brief picture of the algorithm (when it works)

Genetic algorithm glue code does work ... sometimes

(I am so tired of nondeterministic programs…)

Unimpressive runs so far, likely due to a lot of factors - no real selection happening, so optimization is impossible.

TO DO

Make the program WORK!

More mutation/recombination/selection functions.

Could try again with simpler algorithm-simulated annealing?

Better architecture and algorithms - influenced by end-of-term lack-of-time

Rewrite in Stratego and apply general idea to more programs.

Conclusions

This project was too ambitious, at least given my poor planning.

Genetic algorithms are fairly complicated, but very general.

I had never written a non-trivial genetic algorithm or “genetic programming” algorithm before, but it seemed like it would be easy…. It was not - although not for any good reason.

I didn’t figure out how to test smaller parts in any very meaningful way before putting together the algorithm - but I should have.

Haskell was useful for prototyping everything and reasoning about how the parts of the algorithm fit together in the type system - but laziness and an unfamiliar debugging environment made it tedious to get things working.